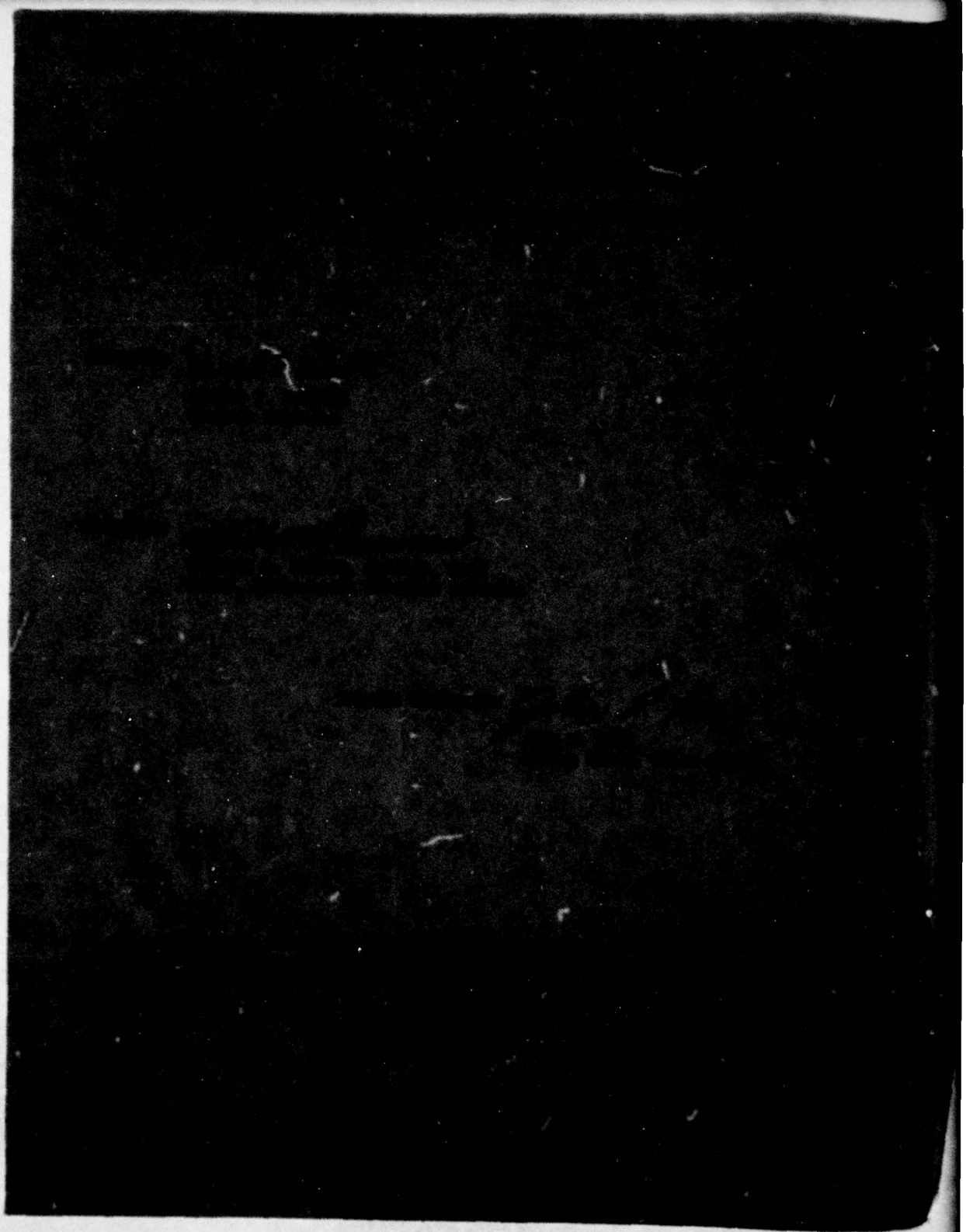
AD-A075 108 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
A JOVIAL VERIFIER.(U)
JUL 79 B ELSPAS , M W GREEN , M S MORICONI F30602-78-C-0031
UNCLASSIFIED SRI-6977-1 RADC-TR-79-195 NL

END
DATE FILMED
11-79
DDC


1·0
1·1
1·25
2·5
2·2
2·0
1·8
1·6
NATIONAL BUREAU OF STANDARDS

D. Elgass M.S. Mariconi
H.W. Braun R.E. Skostak



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-79-195	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A JOVIAL VERIFIER	5. TYPE OF REPORT & PERIOD COVERED: Interim Report. 14 Nov 77 — 31 Dec 78.
6. PERFORMING ORG. REPORT NUMBER: SRI-6977-1	
7. AUTHOR(s): B. Elspas, M.S. Moriconi, M.W. Green, R.E. Shostak	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0231
9. PERFORMING ORGANIZATION NAME AND ADDRESS: SRI International, 333 Ravenswood Avenue, Menlo Park CA 94025	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F, 25320205
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: July 1979
	13. NUMBER OF PAGES: 79
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program verification	Hierarchical design methodology
JOVIAL-J73/I language	Mechanical theorem proving
Proof of correctness	Inductive assertions
Verification/validation	Software tools

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Interim Report describes progress during the first year of research and development on a program verification system supporting the design, development, and formal verification of programs written in JOVIAL-J73-I. The work reported here represents an outgrowth and continuation of earlier efforts concerned with JOVIAL-J3 and JOVIAL-J3(JOCIT version).

Detailed descriptions are presented of progress and future plans in the following areas: (Cont'd on reverse)

DD FORM 1 JAN 73 1473 over

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 281

JOB

a. Analysis of the JOVIAL-J73/I language with respect to verifiability, leading to formal definitions of its syntax and semantics, and to the enhancement of the language by assertion constructs for the purposes of specification and verification;

b. The organization, technical development, and implementation of portions of the JOVIAL Verifier - in particular, of a verification condition generator and deductive sybsystems;

c. The selection, analysis, and verification of two real software systems of significant size and complexity, critical parts of which will be implemented in J73/I and which is hoped to be verified by means of the JOVIAL Verifier.

The report contains two appendices, comprising a formal grammar (in modified BNF form) for J73/I, and a detailed mathematical axiomatization of the semantics of floating point arithmetic for the language.

CONTENTS

ILLUSTRATIONS

I INTRODUCTION

This report describes progress made during the period from 14 November 1977 through 31 December 1978 in developing an effective system for use in developing and maintaining formally verified JOVIAL programs of significant size and complexity. During this period we have:

- Completed a preliminary analysis of the impact of verification concerns on the JOVIAL language.
- Invented techniques for determining the effects of incremental changes to functions, procedures, types, and certain specifications.
- Developed certain useful deduction mechanisms, a parser-generator, and the most important parts of a verification condition generator.
- Carefully selected two real systems, critical parts of which will be verified using the JOVIAL verifier.
- Formulated a comprehensive plan for the future development of the JOVIAL verifier and for verifying certain milestone examples.

Each of these achievements is an important step in attaining our overall goal of producing an effective verifier for JOVIAL programs, and each will be explicated in subsequent sections.

Over the last few years, we have gradually moved away from verifying arbitrary JOVIAL programs, especially those written without attention to formal verification concerns. During that time, we, as well as other researchers in program verification, have come to appreciate the importance of clean program structuring in making formal verification practical. Without a sound methodology for software design and implementation it is difficult, if not actually impossible, even to express adequate formal specifications (assertions) with respect to which correctness is to be proved. The near impossibility of proving "correctness" for large, already existing programs, especially those

written without attention to abstraction, has been often remarked. We have therefore turned our attention to programs developed according to sound methodological principles and expressed in terms of well-structured language constructs. The design and formal specification of JOVIAL programs will be done according to the SRI Hierarchical Development Methodology [22]* (henceforth called HDM). The resulting programs will tend to be well structured, understandable, easily maintainable, and demonstrably consistent with their specifications.

Experience further indicates that the presence or absence of certain programming language features also affects all these desirable properties. Therefore, we performed a careful analysis of each construct in the JOVIAL language, and concluded that nearly all JOVIAL constructs satisfy the requirements of formal verification. We are pleased that only a few restrictions are needed to significantly enhance verifiability for programs in this language.

Even though methodological concerns are addressed and suitable restrictions are placed on the JOVIAL language, the verification activity is too tedious and cumbersome to do by hand. Machine assistance is needed. As a result several program verification systems have been developed [e.g., 15,6,9,10,12,17,20, and 28]. These systems can be viewed as consisting of three main components--a parser for parsing programs and their specifications; a verification condition generator for producing logical formulas (called <u>verification conditions</u>) that, if proved, establish that a program satisfies its specifications; and a proof system for proving verification conditions. Our planned system will also include these functional components, each attuned to the JOVIAL-J73/I programming language [14,18]. Hereafter when the term "JOVIAL" is used in this report the Level-I subset of JOVIAL-J73 should be understood.

In our verifier, as in previous ones, the most complex and least understood of these three components is the proof system. The major issue is the tradeoff between generality and efficiency, which we are

---

* References are listed following Section VI.

addressing by following two complementary lines of research. The first effort is the development of the Boyer-Moore theorem prover [1,5], which is intended for efficient general use. This theorem prover is based on the theory of recursive functions, is axiomatically extendable, and is tailored to proving properties of programs in the sense that it is easy to model (in the Boyer-Moore theory) the inductively constructed objects (e.g., integers and sequences) underlying the semantics of programs. An inescapable fact confronting any such theorem prover is that, for any branch of mathematics that includes inductively defined concepts, there exists no algorithm capable of deciding every question that can be raised. Thus, we are also developing fast, special-purpose theorem provers for certain specialized domains--e.g., for various subclasses of Presburger arithmetic [25,26]. We feel that the marriage of these two approaches to proving theorems will give our proof system a unique and powerful deductive capability.

Other important innovations in our planned system are its support of a formal development methodology and its ability to determine the effects of incremental changes. As an illustration of the latter problem, consider the task of developing a formally verified operating system. A good strategy is first to decompose the system into its functional parts, then to design and verify each part separately. The file system, for example, can be broken down into files and directories, each of whose design and verification involves developing a large, highly interrelated collection of specifications, programs, verification conditions, and proofs. Certainly, numerous revisions--e.g., to correct an error in a program or to augment or reformulate a specification -- would be made in getting this myriad of detailed information to fit together properly. Each revision can raise a variety of complex questions. Suppose, for example, that some specifications and programs dealing with files are changed to allow blocks of file storage to be scattered throughout memory, instead of being allocated sequentially as originally planned. For this example, some of the key questions are:

- Do any previous proofs about files remain valid?
- Does any code for directories need to be recompiled?
- Do all established properties of the file system still hold?
- Is the rest of the operating system affected? If so, how?

These are the kinds of questions the JOVIAL verifier will be able to answer, thereby avoiding excessive redoing of previous work unaffected by incremental changes.

This report is organized as follows. Section II summarizes our analysis of the JOVIAL language, proposing certain enhancements and restrictions for purposes of formal verification. Section III describes our progress in developing the JOVIAL verification system. Section IV outlines our overall technical plan for completing our analysis of the JOVIAL language, for developing the JOVIAL verifier, and for producing milestone examples that reflect our progress. Finally, Section V summarizes our progress to date and indicates our plans for the next year.

II THE JOVIAL-J73/I LANGUAGE

A. The Subset to be Verified

We have carried out an extensive review of the Level-I subset of JOVIAL-J73 as it is described in [14] and [18]. These reference documents were used to construct a formal BNF grammar for use by the verification system. The grammar appears in Appendix A to this report. No subsetting was employed in its construction--i.e., all of the syntactic features described in [14] were retained in our grammar. Consequently, when our parser/transducer based on this grammar has been completed it should be able to parse any legal J73/I program into an internal form acceptable to our verification condition generator (VCG) described below. In fact, our grammar also provides for assertion constructs (see Section II-C) needed specially for verification. Thus, this grammar is actually an extension to standard J73/I.

The semantics of J73/I, however, posed a more difficult set of questions for verification. One reason for this is that they are defined only informally by the reference language documentation [18]. Wherever any reasonable doubt occurred as to the intended semantics, we wrote sample J73/I programs to resolve such possible ambiguities. In all, several dozen such small programs were compiled and executed on our DEC-10 computer, in addition to a good many more that were tested earlier in the effort simply to gain experience and familiarity with the language. A number of these programs were concerned with numeric semantics, leading to an axiomatization that appears in Appendix B. Perhaps the greatest concerns were occasioned by the mechanics of passing input- and output-parameters, the side effects on global variables, the initialization of formal parameters, and the handling of name scopes in procedures. Such concerns were resolved by the aforementioned sample programs in conjunction with careful attention to the documentation.

The conclusion of this analysis of JOVIAL syntax and semantics was that with the exception of the few excluded constructs discussed below, our verification system could be designed to handle all of J73/I. We have probably been conservative in this assessment, since some of the excluded features could probably be accommodated, though with some difficulty, and leading to greatly increased time spent in deduction as well as in preprocessing.

B. Excluded Constructs

One of the most serious pitfalls for formal verification of programs lies in the possibilities afforded for "aliasing" in some languages, including JOVIAL. Aliasing occurs whenever there are two or more pointers to the same data item in physical memory. Assignments to any one of these pointers will then result in modification of the object pointed to by all the other pointers. Moreover, in J73/I such assignments need not be made through explicit assignment, but can also occur through the passing of actual output parameters by reference. The difficulty such aliasing poses for verification is that the VCs generated for such programs must then contain large conditional expressions over all pointer variables that may potentially be aliased with the one actually subjected to assignment. The detection of which pointers potentially share structure may require subtle analysis (perhaps during a preprocess phase of verification condition generation), and the actual proof of the large resulting VCs can be extremely tedious. Whether two pointers are actually (as opposed to potentially) aliased may depend on the run-time environment and, therefore, require even more subtle proof techniques.

A particularly troublesome form of aliasing can arise through the use of table and block overlays. This is because the overlapping of memory references is thereby made dependent on the calculation of absolute memory locations, something that should be avoided at all costs in formal verification.

It is, therefore, considered necessary to exclude from consideration, at least initially, any features of JOVIAL that could lead to aliasing. The impact of this restriction on the flexibility and utility of JOVIAL programs is not entirely clear at this point. There will certainly be an increase in the data storage requirements for such programs, e.g., if overlays are prohibited. However, we are attempting to find ways around this problem, and it may not be necessary to forbid all kinds of aliasing, but merely to "tame" it.

Another troublesome aspect of real programming languages lies in the possibility of generating hidden side effects through the evaluation of expressions, particularly in function calls. The insidious nature of such effects may be well appreciated through the observation that, too often, overly "clever" programmers will make deliberate use of such side effects--e.g., in function calls occurring within Boolean tests--without adequately appreciating that an optimizing compiler may change the order of evaluation of subexpressions and thereby destroy or distort the intended effect. It is impossible to account adequately for such compiler optimization effects without a formal specification of what the compiler might do. (Note, for example, that in several places the reference document [14] states that the order of evaluation of certain expressions must be regarded as undefined.) In view of such uncertainties, we have had to restrict the use of (expression level) function calls to functions without side effects on global variables. Fortunately, this does not rule out procedures (callable at statement level). These are allowed to have side effects. Moreover, in the absence of aliasing, the presence of side effects in formal functional procedures can be detected by straightforward analysis of the function body. We might therefore even permit such procedures to be submitted to the verifier, but place the burden of guaranteeing the absence of any hidden side effects on the programmer, rather than on the verifier. On the whole, however, it is probably the better course simply to rule out assignments to external variables within a function body, whether by explicit assignment or by calls to procedures invoking side effects. This restriction on the programming style of the JOVIAL programmer is

not a severe one. We regard its impact on the programmer as minimal. Moreover, most programmers would probably agree that the restriction is good programming practice and that it facilitates readability, checkability, and maintainability of the resulting code.

A final exclusion concerns the DEFINE construct. This is essentially a source macro facility that permits the creation and modification of source code by the substitution of parametrized character strings. The verification of J73 programs that use the DEFINE feature would require the deductive system to be able to deal with character string substitutions (often through several levels, as where a DEFINE call uses parameters involving a "defined" name, or a DEFINE string contains such a name). Fortunately, one can simply apply the verification system to the expanded code. The !LIST DEFINE directive of J73/I provides the capability for performing this macro-expansion, and we plan to have the verifier perform its analyses on the expanded code from which all DEFINEs have been removed. Note that this is not really a restriction on the source language, but rather a question of the philosophy of verification. Since it is the expanded code that gets compiled, verification of the expanded code is actually a sounder approach than verification of the source code. If it were feasible to verify compiled machine code (we believe it is not!) it would be sounder to do so than to verify source code, since the compiler semantics would then not come into question. Macro-expansion, however, is a small step in the same direction, and we believe that macro-expanded code can be verified.

C. Enhancements for Verification

Certain additions to the JOVIAL syntax must be provided for the purpose of verification. We refer here to the entry-, exit-, and intermediate-assertions required in the Floyd-Hoare approach, and also to specifications (in the style of HDM) for hierarchical verification. We chose to add intermediate assertions to the formal syntax by including an additional production <assert_stat> under the nonterminal <simple_stat> (which is, ultimately, one of the forms of the nonterminal

<statement>; see productions /*298*/, /*312*/, and /*324*/ in the grammar shown in Appendix A.)

The statement type <assert_stat> (see Appendix A, production /*378*/) can take any one of the three forms:

ASSERT formula ';'
ASSUME formula ';'
PROVE formula ';'

The minor semantic differences among these three kinds of assertions are discussed in Section III-B-3. The first type, ASSERT formula, is the usual embedded Floyd assertion; the other two types are variants useful for providing flexibility in the forms of the theorems produced by the verification condition generator (VCG). These statements are intended only for use by the VCG and are supposed to be ignored by the actual J73/I compiler. J73/I programs legal by our grammar may contain such assertion statements wherever a statement is legal in standard J73/I. For such programs to be accepted by a standard J73/I compiler, the assertions must, of course, be eliminated or made transparent once the program has been verified. One possibility is to have a preprocessor transform all such assertions into comments before submitting the program to the compiler.

In addition, it was convenient to permit inductive assertions to be embedded directly in iterative statements, i.e., any of the J73/I loop_statement forms. This was accomplished by allowing an optional clause of the type assert_stat within both while_stat and for_stat (see Appendix A, /*363*/ and /*364*/). Any J73/I iterative statement lacking such an inductive assertion is presently transduced with a vacuous assertion clause (ASSERT T) for use by the VCG. Thus, it is also possible for the human verifier to insert loop assertions manually after parsing/transduction if he wishes. However, this possibility should be considered merely as an interim convenience for use while the system is under development. In our completed verifier the system will detect the absence of any such essential assertions and will insist that the user provide them in the source code before transduction and the generation of verification conditions can be completed.

Procedure-level input/output specifications have been provided by allowing optional entry/exit assertions (as declarations) within the body of a procedure declaration. See Appendix A, items /*263*/, /*296*/, /*301*/, and /*311*/. The syntax for the added declaration type <assert_decl> is:

```
ASSERTIN formula ';'
ASSERTOUT formula ';'
```

The key words ASSERTIN and ASSERTOUT were chosen so as not to conflict with the JOVIAL reserved words ENTRY and EXIT, which would otherwise have been the most natural terms to use for the entry and exit assertions associated with procedures. These assertion declarations form part of the mechanism for hierarchical verification of programs that contain procedure calls. The declarations are processed only by the VCG subfunctions concerned with procedures. The reason for including ASSERTIN and ASSERTOUT declarations in addition to the above ASSERT statements is to facilitate structured proofs of program correctness. This approach allows the verification of the body of a formal procedure to be decoupled from any calls to that procedure. The formal procedure need then be verified only once (corresponding to its declaration) regardless of how often it may be invoked in a program. More detailed discussions of this important point may be found in [9] and [10].

Future additions of similar nature may be required to accommodate HDM specifications, although it is also possible to handle such specifications apart from the JOVIAL implementation itself. JOVIAL programs would then first be specified (e.g., in the HDM language SPECIAL) through a succession of HDM modules at several levels of abstraction. However, the SPECIAL language would then need to be augmented to accommodate the JOVIAL control constructs required in the implementation of these modules.

III THE JOVIAL VERIFIER

The gross architecture of the JOVIAL verifier is shown in Figure III-1. The human designer/verifier communicates with the assistant, which invokes the translator for parsing and type checking, the verification condition generator for producing verification conditions, and the proof system for attempting a mathematical proof of certain formulas. This section attempts to describe the main functional characteristics of each of these components, its general organization, the progress made in its development, and what remains to be done. Specific technical details are generally omitted if they have already been described in the technical literature. Such relevant documents are cited in the text.

An additional fact that should be mentioned is that some system components (especially the proof system) have been supported in part by other projects in the Computer Science Laboratory. It has become apparent that this relationship has been mutually beneficial for all the projects involved.

A. Translator

The JOVIAL verifier must contain two translators--one for SPECIAL [24] (the formal specification language to be used), and one for the dialect of JOVIAL to be supported. Each translator will consist of a parser, generated by the INTERPG parser generator [27], and a type checker. Their output will be a new representation of the source code appropriate for internal manipulation by the verifier.

An important consideration in the design of this internal representation is that it should be able to accommodate any JOVIAL dialect, as well as other source languages such as DOD/1. Attaining this goal is important because a JOVIAL verifier based on such an

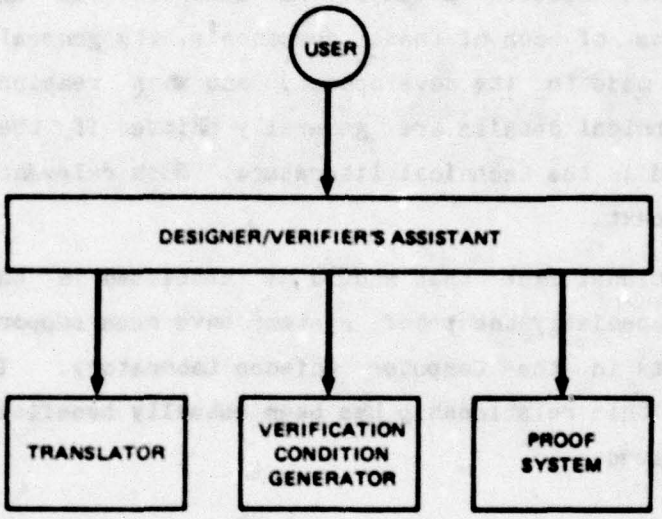


FIGURE III-1 GROSS ARCHITECTURE OF THE JOVIAL VERIFIER

internal form can be fairly directly converted to support an entirely different source language. The translator must be rewritten, but all the rest of the verifier should remain essentially unchanged. Our initial cut at designing such an internal form adhered to the following desiderata:

- The internal form must have a simple and well defined semantics.
- The verification conditions generated by the verification condition generator (operating on the internal form translated from some source language program) should be equivalent to those that would be generated by a verification condition generator operating directly on the source language program.
- The internal form should be minimal, having only constructs necessary for the attainment of the other goals.
- The internal form should be applicable to a wide variety of source languages, but should not be unnecessarily predisposed toward any single source language or group of source languages.
- It must be possible to associate statements and constructs in the internal form with the statements and constructs in the source language program from which they were derived.

Certain simplifying restrictions were made in order to expedite the development of this preliminary version. These restrictions will be relaxed in later versions of the internal form. Presently, the restrictions are as follows:

- Only sequential programs are considered. We attempted to avoid doing anything that would make parallel programs difficult to handle, but we did not investigate the problems of handling concurrency on the internal form.
- The source languages considered are JOVIAL, MODULA, PASCAL, and DOD/1 (actually verifiable subsets of these languages). Again we tried to avoid doing anything that would preclude handling of other languages but we did not explicitly consider any other languages.
- Parameter aliasing is not permitted.

The internal form is heavily based on the assembly-language-like language developed by R. Boyer and J S. Moore as part of their formal definition of HDM [4]. The first cut at the internal form grammar is given below:

```
<function implementation>
    ::= (<function name> (<arg list>) <inst seq>)

<arg list>
    ::= <null string> | <arg> <arg list>

<inst seq>
    ::= <null string> | <instruction> <inst seq>

<instruction>
    ::= <label> <instruction>
      | (GO <label>)
      | (ASSERT <exp>)
      | (COMMENT <quoted string>)
      | (RETURN <exp> <case list>)
      | <exp>

<case list>
    ::= <null string> | <case> <case list>

<case>
    ::= (<literal exp> <inst seq>)

<exp>
    ::= <var>
      | (<function name> (<exp list>) <case list>)

<exp list>
    ::= <null string> | <exp> <exp list>
```

The parser/transducer serves to define the source language syntax (more properly, the context-free portion of that syntax) to the verifier. For purposes of program verification it may be assumed that all programs will first be checked for legality by an actual J73/I compiler (for the intended machine). Hence, it is redundant to incorporate checking for the context-sensitive syntax in the front end of the verifier. (In fact, it might be argued that the syntactic checking carried out by the parser/transducer is also redundant. However, this checking is a by-product of parsing the program into internal form, and it certainly does no harm.) Examples of syntactic checks <u>not</u> carried out by our parser/transducer are:

- Scope checking of declaration before use for names (of variables, procedures, items, tables, etc.)

- Type checking of actual procedure parameters against the corresponding formal types
- Multiple definition of names in a given scope
- Return statement occurring outside a procedure
- Multiply-defined switch points.

On the other hand, checks of the following illegal situations (among others not listed) are carried out by our parser/transducer:

- Misuse of a reserved JOVIAL word as an identifier
- Output parameters in a function call or function declaration
- Appearance of constructs at inappropriate contexts (e.g., use of a function call at statement level)

These checks are made even though the same matters would be checked by the actual JOVIAL compiler.

B. Verification Condition Generator

1. General

The purpose of the verification condition generator (VCG) in a software verification system is to generate mathematical formulas that express conditions for consistency between a program and its specifications. These formulas are called verification conditions (VCs). Our VCG will operate on internal representations of the JOVIAL programs to be verified. To do its job properly the VCG must incorporate knowledge about the semantics of the input language (as reflected in the internal representation). The internal representations are produced by an input parser/transducer mechanically constructed from a formal (modified BNF) grammar. A tentative grammar for JOVIAL-J73/I is shown in Appendix A. The combination of the parser/transducer with the VCG, which we refer to as the verifier "front end", is, in fact, a verification condition compiler for the actual source language (J73/I).

Our plan is to complete the present VCG implementation described in the next subsection to provide us (by the end of this year) with a theorem generator for a prototype JOVIAL verifier capable of

verifying J73/I programs that contain limited procedural hierarchy. This first-cut system would, however, not incorporate the HDM methodology. Our final, delivered system will employ the approach described in Section III-A, using secondary translations of the initial internal forms, in order to incorporate HDM, and with it data abstraction.

2. Internal Forms

The internal forms produced by our parser/transducer will be essentially faithful representations (as parse tree structures) of the input source language program. There are two slightly different ways in which these initial internal forms can be processed to generate verification conditions. The direct approach is to implement a VCG (in terms of predicate transformer functions) that operates directly on these initial internal forms. This is the approach that was followed in the preceding RPE/1 and RPE/2 projects for JOVIAL-J3 [9,10]. In our first year's work under the present program VCG implementation has also largely pursued this philosophy. The second approach entails a two-step process: (1) translating the initial parsed forms into a low-level, assembly-like language (see Section III-A), and (2) implementing either a conventional VCG or a symbolic interpreter for this language. This second approach is likely to be simpler in the long run and also has the advantage of providing a more uniform mechanism which could be applied to other source languages. Thus, the specialization to a particular source language (such as JOVIAL-J73/I) appears entirely in the internal form translator, and a single VCG may be shared among several languages. There is an additional advantage to this approach: From some early experiments in verifying consistency for modules specified in terms of the HDM language SPECIAL [22], it appears that this approach is better suited for interfacing conventional verification with hierarchical schemes such as HDM. In these experiments the low-level language was given operational definition by means of a symbolic interpreter. However, an axiom-based VCG could probably be employed in much the same manner.

The implementation described in the next subsection is based on the first (direct) approach.

3. Present VCG Implementation

The implementation described briefly here is one based on the notion of predicate transformers [7]. It currently handles only a subset of the executable statement types of JOVIAL-J73/I. The input expected by this VCG subsystem is the initial internal form of a JOVIAL program annotated by inductive assertions and provided with input and output specifications for any embedded procedure declarations. An input assertion and an output assertion for the program as a whole are also assumed to have been provided. The internal (loop-cutting) inductive assertions are required at each targeted labeled statement and within any loop statements (see Section II-C).

The VCG output is a list of VCs, one for each executable path between assertion points. Each such VC is computed by "pushing backward" an assertion from a "final" assertion point until a prior assertion point is reached. The transformation through each type of executable statement is performed by a VC subfunction that implements a Hoare axiom or rule [13] for that particular statement type. Declarations (of data items and procedures) are transparent to this process of predicate transformation, but they produce a side effect on the data base of the VCG subsystem that is analogous to the production of the symbol table by a conventional compiler. The whole process is complete when the program's input assertion has been reached.

The main predicate transformer function is called VCS. The subfunctions referred to above have names of the form VCR:IF, VCR:ASST, VCR:ASSERT, and the like, each one designed to produce the required predicate transformation on its second input argument when the statement being processed is an if-statement, assignment statement, or assert statement, respectively. The particular VCR subfunction appropriate to each statement type of JOVIAL is automatically invoked by VCS through examination of the internal form of the statement being processed. This

selection is facilitated by the fact that the initial element of each internal form is a key word (e.g., WHILE, IF, FOR, and :=) that uniquely identifies the statement type.

As discussed in Section II-C, the machinery of verification required the addition of assertion statements to the JOVIAL syntax. In fact, the assertion types ASSERT, ASSUME, and PROVE are subsumed under the nonterminal <assert_stat> (see Appendix A). The single assertion-type ASSERT would actually have sufficed, but it is technically convenient to have all three available. They differ as to the details of the verification conditions (VCs) they induce VCG to produce, in particular, with regard to number and complexity. Roughly speaking, the differences are that (1) an "asserted" predicate must be proved when reached, and it also initiates a new VC path; (2) an "assumed" predicate need not be so proved, and does not create a new path, but is instead combined with prior predicate information; and, finally, (3) "prove" predicates must be proved, but they are like "assumed" predicates in not starting new paths. The use of PROVE in place of ASSERT for intermediate assertions will therefore result in fewer, but more complex VCs.

The ASSERT statement is the usual embedded Floyd assertion. When one is encountered by VCG in scanning the program, VCG begins to construct a logical formula (VC) with the asserted predicate as hypothesis. This VC is not completed until the next ASSERT statement is encountered, at which point (a modified form of) this terminating assertion is used as the conclusion for the constructed formula. The details of the modification are determined by the code intervening between the two assertions. If an ASSUME-type assertion appears between the initial and terminal ASSERTs, the "assumed" predicate would be conjoined to the hypothesis. Since ASSUMEd predicates are not subjected to proof (they only appear in hypotheses), the ASSUME statement would normally be used only for initial assertions of a program. However, it might also be used for intermediate assertions that are subjected to some independent check (e.g., run-time checks by compiler-generated code).

The purpose of the PROVE statement is to allow introduction of an intermediate assertion (which must be proved when it is reached) but which also picks up all prior hypotheses for that path. This has the advantage that predicates that are true globally over a program do not need to be repeated (as would be the case if ASSERT were used instead of PROVE).

A precise description of the above distinctions in terms of Hoare axioms for the three types of assertion statements appears in an SRI report [11]. The same report also contains formal proofs of consistency between the Hoare axioms and a LISP implementation (for a much simpler VCG).

The types of executable JOVIAL statements for which versions of the corresponding VCR subfunctions have been implemented are:

- Assignment statements (simple and multiple left sides)
- Conditional statement
- Loop statement (the "while" form)
- BEGIN...END statement sequences
- GOTO statement
- Assertion statements (assert, assume, and prove)
- Procedure calls (with some restrictions on side effects)
- Function calls (without side effects)
- STOP statement.

Of the types listed above, the procedure call poses the greatest difficulties for VCG implementation, and also the worst potential pitfalls in "getting the semantics right". We have used a version of the procedure call rule given for EUCLID [16], with certain modifications necessitated by differences in the semantics of parameter passing for structured variables in JOVIAL. The present VCG implementation (for procedure call) cannot be considered final, since JOVIAL has some idiosyncracies that probably have no counterparts in other languages--e.g., formal output parameters that are not also input parameters are given a default initialization (to zero) on procedure entry.

Among the main features of JOVIAL that have not been accommodated as yet in the present VCG implementation are the following:

- Switch statements
- The for-loop statement
- Return statements within procedures
- Intrinsic functions
- The semantics of real machine arithmetic.

None of these pose really difficult problems, except perhaps the last item. Even there the difficult portion of the job has already been done--viz., a mathematical axiomatization of most of these semantics (see Appendix B). The actual implementation of this axiomatization of floating-point arithmetic remains to be carried out. The other items above can be handled by a combination of modifications to the VCG developed for J3-JOCIT [10] and the judicious use of mappings (secondary transductions within the VCG) of these statement types into types already handled--e.g., the for-loop statement can be internally transduced into an equivalent while-loop for which the VCR subfunction exists. The intrinsic functions and return statement will, however, require special handling.

C. Proof System

Since an effective proof system is essential to mechanical program verification, considerable effort has been spent in exploring various approaches to proving theorems. Attention was primarily focused on the Boyer-Moore recursive function theorem prover [1,5], on improving an earlier implementation of Smullyan's analytic tableaux [10], and developing fast decision procedures for certain classes of formulas that frequently occur in program verification [25,26]. Each of these deductive mechanisms is explained in detail in the references cited; consequently, specifics will not be given here.

After developing, implementing, and extensively testing these three approaches to deduction, we have decided to use the Boyer-Moore theorem prover as the general deductive mechanism and to implement special-

purpose provers for the specialized theories for which fast decision procedures have been devised. The semantic tableaux system was eliminated from consideration because it generally appeared to be very tedious to use and because its achievements pale in comparison to those of the Boyer-Moore prover. A principal reason for the latter comparison is that the tableaux system user was required to supply instantiations manually when applying previously proved lemmas. The Boyer-Moore prover, on the other hand, contains an efficient pattern-matching mechanism to perform the corresponding task automatically. We plan to invoke the fast, special-purpose decision mechanisms from within the Boyer-Moore system, either automatically or under user control, where appropriate. Ultimately, such efficient mechanisms may be incorporated directly into the Boyer-Moore package.

1. The Boyer-Moore Theorem Prover

The Boyer-Moore prover has been used to prove an impressive list of difficult recursive programs (written in the Boyer-Moore theory), including a simple optimizing expression compiler [2], an expression parser, a fast string-searching algorithm [3], a mechanical theorem prover for propositional calculus, and an arithmetic simplifier. It has also proved a variety of mathematical theorems, the deepest being the unique prime factorization theorem (which is derived solely from the Peano axioms for integers and lists and previously proved theorems).

We are planning several extensions to the Boyer-Moore prover that will:

- Make it easier to state and prove many kinds of assertions (e.g., by providing recursive, schematic concepts)
- Ease the burden on the user during proofs (e.g., by mechanically inventing lemmas)
- Increase the prover's competence in many important domains (e.g., graphs, the arithmetic of the rational numbers, and combinatorics).

Specific investigations (supported under other contracts) are being carried out in each of these directions, but their description is necessarily highly technical and is beyond the scope of this report.

2. A Simplifier for Specialized Theories

Prior to the period covered in this report, much emphasis was placed on efficient mechanical deductive techniques for specific formula domains. This emphasis was motivated in large part by a need for fast, automatic, decision mechanisms in our first system for verifying JOCIT programs. Previously, most deductive systems designed specifically for program verification applications were of the heuristic, goal-driven type. The first SRI Program Verifier [8] and a substantial part of the RPE/1 system [9] were subgoaling systems that depended strongly on ad hoc heuristics. While these systems were quite general and easy to modify, they tended to be unreliable, incomplete, and often too slow to handle many of the larger, more complex verification conditions in a reasonable period of time. Our early experience, dating back to the RPE/1 project (1975-76) and before, indicated that a substantial fraction of the formulas actually encountered fall within classes (so-called decidable domains) for which validity is algorithmically decidable (i.e., without recourse to heuristic techniques). Observe that a decision procedure (when applicable) is preferable to a procedure that merely proves validity (a proof procedure). A decision procedure will (for any formula in its domain of competence) establish whether that formula is valid or not valid. Proof procedures, on the other hand, can suffice to establish validity for some formulas (over a domain), but the results are inconclusive when the proof procedure happens to fail on a particular formula. Such procedures are therefore often called semidecision procedures. The problem is, of course, that decision procedures are known to be theoretically impossible for many important domains (e.g., even for number theory--the arithmetic of the integers under addition, subtraction, and multiplication). Nevertheless, there are simpler formula domains for which decision procedures are known to exist. Perhaps the most important of these domains is that of Presburger arithmetic*. Certain extensions of this

---

* Roughly speaking, Presburger arithmetic consists of logically quantified formulas with integer constants and variables where only addition, subtraction, and multiplication by constants are allowed.

domain and also some of its subdomains are of considerable relevance to program verification. Accordingly, we initiated a course of research with the aim of producing fast, non-heuristic algorithms for these classes that could be implemented and tested in the RPE/2 system.

That effort resulted in fast decision procedures for (1) quantifier-free* Presburger arithmetic [26] augmented by uninterpreted function and predicate symbols, (2) a class of unquantified equality formulas with function symbols [25], and (3) deciding the "feasibility" of sets of linear inequalities over the integers or the rationals**.

While this work on fast decision procedures enabled us to prove automatically many of the verification conditions and fragments of verification conditions encountered in the RADC efforts, it by no means facilitated automatic proof of all of the formulas we came across.

Part of the inadequacy was simply a matter of speed. Certain of the formulas that we encountered had Boolean structure that produced a combinatorial explosion too large to handle with a reasonable limitation on CPU time. A more serious problem was simply lack of generality. First of all, the immediate application of our procedures was limited to quantifier-free formulas--i.e., formulas, which, when placed in a certain canonical form (prenex normal form), contain no occurrences of existential quantifiers. Many of the verification conditions that arose contained such quantifiers, but were otherwise similar to formulas in the classes we treated. Secondly, the system lacked the generality needed to deal with the semantics of a number of commonly occurring mathematical operators, such as multiplication, division, and set manipulation.

---

* I.e., excluding the use of the logical quantifiers "forall" and "there exists".

** A set of arithmetic inequalities is said to <u>feasible</u> (or satisfiable) over a domain D if for each of the free variables appearing in the inequalities values in D can be found for which all the inequalities are true.

The inability to handle such constructs was a direct result of the decision to stay within the bounds of the decidable. The theories for which we implemented decision procedures are, in a sense, dangerously close to that boundary. For example, extending the theory of unquantified Presburger formulas with function symbols (which is decidable) to include existential quantification results in undecidability. Likewise, augmenting (arbitrarily quantified) Presburger arithmetic with even a single uninterpreted predicate symbol also takes one outside the bounds of decidability. The theory of integer multiplication, and the unquantified theory of primitive recursive functions are, of course, also undecidable. Nevertheless, we are optimistic about the possibility of devising fast decision procedures for other useful domains (e.g., the theory of "bags" or multisets--sets with repeated elements) that will have a significant impact in proving verification conditions.

We are planning to continue these investigations into discovering fast decision procedures for such relevant decidable theories, while at the same time implementing a unified simplifier for applying them.

3. Future Plans

We are currently exploring the feasibility of integrating the Boyer-Moore prover with the more specialized simplifier. If this integration proves to be inappropriate, the current plan is to use the simplifier as a preprocessor for the Boyer-Moore prover. That is, verification conditions will be simplified (thereby removing much of the clutter typically contained in verification conditions) before a proof is attempted by the more general Boyer-Moore prover. A second possibility would be to use simplification during the process of verification condition generation, as in some earlier verifiers [6,20].

Another interesting idea now being explored concerns the simplifier's ability to detect invalid formulas. An invalid verification condition can arise because either the program being

proved, its specifications, or both are "wrong" and therefore in need of change. The question is, How can counterexamples generated by the simplifier be effectively used by the designer/verifier's assistant (described in the next section) to show the user what must be changed and what may be affected by any such change? Such information would be of inestimable value in debugging a program and its specifications.

In summary, we have settled on an approach to theorem proving, have made significant advances in two complementary directions, have plans for further advances, and are in the process of deciding the exact structural organization of the proof system.

D. Designer/Verifier's Assistant

Developing and maintaining formally verified programs, especially large ones, is an incremental activity. Specifications, programs, and proofs are gradually built up and frequently revised. Consequently, one is faced not only with the problem of constructing these data, but also with the complex problem of determining the effects of incremental changes to it. The previously described components of the verifier assist the user in constructing the data, while the designer/verifier's assistant assumes the responsibility for reasoning about incremental changes. Its primary functions are to answer "what" and "why" questions about the effects of hypothesized changes and to integrate actual changes into an existing development in a manner that keeps intact previous work that remains valid. This capability usually saves large amounts of costly and unnecessary reprocessing.

1. Reasoning About Incremental Changes

The assistant embodies a unified theory of how to reason about incremental changes to a design or verification. For each development, the assistant builds a detailed model containing information about key parts of the program's design and verification and their relationships. It then applies its knowledge--the same kind of knowledge an expert would have--to integrate new or changed information into the existing

model, always keeping intact still-valid previous work. For example, if the specification of a function is changed, the assistant deduces what parts of the verification are unaffected.

2. Explaining the Effects of Hypothesized Changes

One of the most difficult problems in designing and verifying programs is trying to understand the effects of a change. Before changing the type of an argument in the parameter list of a function, for example, it helps to know what programs or specifications could have type conflicts, what part of the verification could be invalidated, why it could be invalidated, and so forth. The purpose of the assistant's explanation facility is to make this kind of information available in order to give the user a general understanding of the potential effects of revisions before they are actually made.

All "what" and "why" questions answered by the explanation facility are variants of a few basic kinds of questions. Several samples are given below:

- What are the effects of completing the definition of function X?
- What are the effects of changing its exit assertion?
- What are the effects of changing the parameter list?
- What are the effects of changing X's body?
- What is affected if lemma X is modified?
- What f is affected by altering type definition X?
- Why is Y affected?

Answers to these questions vary in form and content according to context. The questions are understood using a simple pattern-matching scheme, that works well for the limited domain of discourse. The assistant then uses its theory of incremental changes to deduce what could be affected if the indicated change were made. It is straightforward to then format the answer for English output.

3. Current Status and Future Plans

A working prototype of a designer/verifier's assistant was developed in the recent Ph.D. thesis of Moriconi [20]. Effort over the past year has been spent on investigating the principles which underlie this original prototype. We have successfully identified and explained the fundamental ideas behind this work and described our results [19]. With this increased understanding, we can now proceed to develop such a facility for the JOVIAL verifier.

There are two complementary directions our future work in this area will follow. First, it is time to develop a formal mathematical basis for the (currently informal) theory of how to reason about incremental changes. We feel that this is possible because of the experience gained from our previous investigations and the prototype implementation, both of which support the validity of the key ideas. This formalization is necessary if we are to attain the ultimate goal of providing a formal basis for the entire JOVIAL verifier. Second, we will apply the ideas described in [19] to accommodate JOVIAL and the SRI Hierarchical Development Methodology. This will result in a working computer program capable of determining the effects of incremental changes and of answering questions about the effects of hypothesized changes to any JOVIAL program developed using our verifier.

IV MILESTONE EXAMPLES

We are considering several programs for use in demonstrating the utility of the JOVIAL verifier. The programs verified in previous systems were characteristically small in size, textbook or toy examples. They include certain array sorting and rearrangement programs; algebraic computation programs, such as square root, quotient, factorial, and exponentiation; and tree-searching, pattern-matching, and unification programs. Our goal is to formally specify and verify a real software component that is intended for actual use.

We have tentatively decided to verify certain (as yet undetermined) components from two major systems currently being developed in the Computer Science Laboratory at SRI International. The first system, called SIFT (Software Implemented Fault-Tolerance) [29], is an ultrareliable computer for critical aircraft control applications that achieves fault tolerance by the replication of tasks among processing units. The second system, called PSOS (Provably Secure Operating System) [21], is a secure capability-based operating system. Both SIFT and PSOS are being developed under other projects in the Computer Science Laboratory and are in various stages of design and implementation. The discussion below gives a brief overview of each of these systems. For more detailed descriptions the reader should refer to [29].

A. Fault-Tolerant Avionics Computer

Modern commercial jet transports use computers to carry out many functions, such as navigation, stability augmentation, flight control, and system monitoring. Although these computers provide great benefits in the operation of the aircraft, they are not critical. If a computer fails, it is always possible for the aircrew to assume its function, or

for the function to be abandoned. (This may require significant changes, such as diversion to an alternative destination.) NASA, in its Aircraft Energy Efficiency Program, is currently studying the design of new types of aircraft to reduce fuel consumption. Such aircraft will operate with greatly reduced stability margins, which means that the safety of the flight will depend upon active controls derived from computer outputs. Computers for this application must have a reliability that is comparable with other parts of the aircraft. The frequently quoted reliability requirement is that the probability of failure should be less than 10^{-9} per hour in a flight of ten hours duration. This reliablility requirement is similar to that demanded for manned space-flight systems.

As the name "Software Implemented Fault Tolerance" implies, the central concept of SIFT is that fault tolerance is accomplished as much as possible by programs rather than hardware. This includes error detection and correction, diagnosis, reconfiguration, and the prevention of a faulty unit from having an adverse effect on the system as a whole.

The structure of the SIFT hardware is shown in Figure IV-1. Computing is carried out by the main processors. Each processor's results are stored in a main memory that is uniquely associated with the processor. A processor and its memory are connected by a conventional high-bandwidth connection. The I/O processors and memories are structurally similar to the main processors and memories, but are of much smaller computational and memory capacity. They connect to the input and output units of the system, which, for this application, are the sensors and actuators of the aircraft.

The SIFT system executes a set of tasks, each of which consists of a sequence of iterations. The input data to each iteration of a task consist of the output data produced by the previous iteration of some collection of tasks (which may include the task itself). The input and output of the entire system is accomplished by tasks executed in the I/O processors. Reliability is achieved by having each iteration of a task executed independently by a number of processors. After executing the

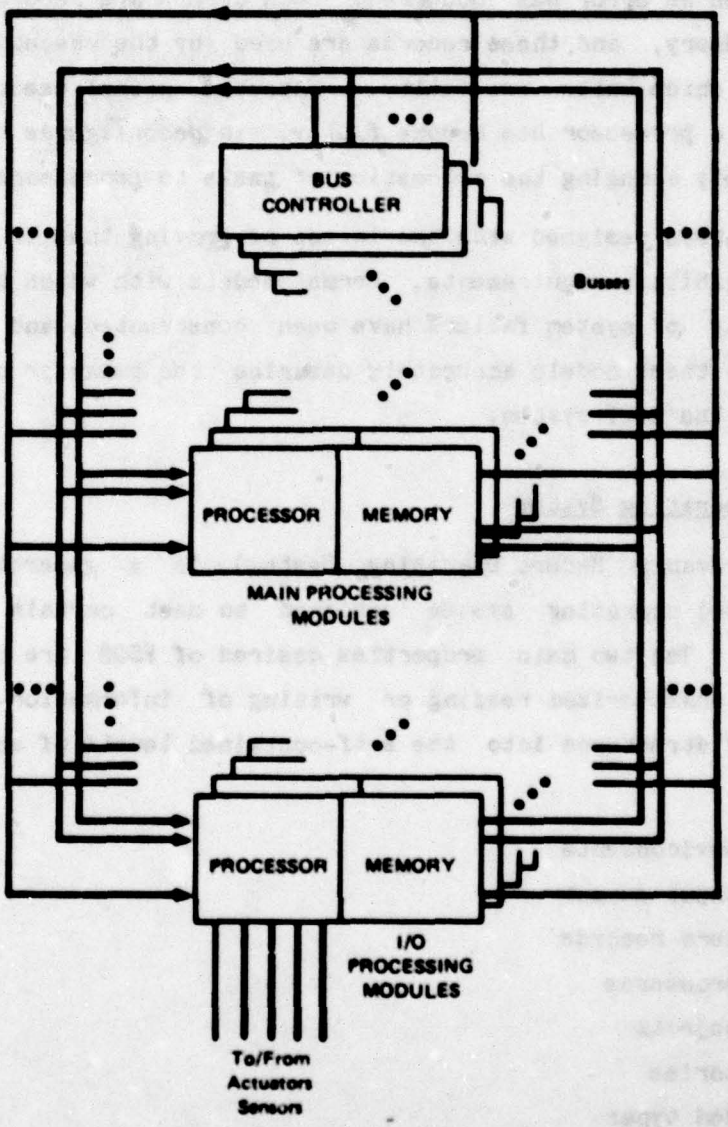


FIGURE IV-1 STRUCTURE OF THE SIFT SYSTEM

iteration, a processor places the iteration's output in the memory associated with the processor. A processor that uses the output of this iteration determines its value by examining the output generated by each processor that executed the iteration. Typically, the value is chosen by a "two out of three" vote. If all copies of the output are not identical, then an error has occurred. Such errors are recorded in the processor's memory, and these records are used by the executive system to determine which units are faulty. When the global executive has decided that a processor has become faulty, it reconfigures the system by appropriately changing the allocation of tasks to processors.

SIFT has been designed with the intent of proving that it meets its stringent reliability requirements. Formal models with which to analyze the probability of system failure have been constructed, and we intend to prove that these models accurately describe the behavior of certain components of the SIFT system.

B. Secure Operating System

PSOS (Provably Secure Operating System) is a general-purpose, capability-based operating system designed to meet certain security requirements. The two main properties desired of PSOS are that there shall be no unauthorized reading or writing of information. PSOS is hierarchically structured into the self-contained levels of abstraction shown below:

- User environments
- User input-output
- Procedure records
- User processes
- User objects
- Directories
- Extended types
- Virtual memory (segmentation)
- Paging
- System processes and input-output
- Basic operations (e.g., arithmetic)

- Real memory
- Capabilities and interrupts.

Each of these levels provides an abstraction useful in the implementation of the operating system. The plan is to verify at least one of the critical levels of this hierarchy.

V PLAN FOR FUTURE DEVELOPMENT OF THE JOVIAL VERIFIER

This section outlines the overall research and development plan for the remainder of the current RADC contract, covering the period from 1 January 1979 through 31 March 1981. The plan consists of a list of the major remaining tasks and a schedule for their completion. This plan is summarized by the chart of Figure V-1 and is explained below.

A. Analysis of JOVIAL

We will complete the identification of a verifiable JOVIAL subset and exhibit a formal semantics for it. This semantic definition will be formulated as either Hoare-style axioms for the JOVIAL source language, or equivalence-preserving translation rules from JOVIAL to a formally defined internal form.

B. A Sample Scenario

We will construct (mostly by hand) a fairly detailed sample scenario that illustrates the kinds of things we intend to be embodied in the JOVIAL verifier. This scenario is important in that it will guide the development of the entire verifier and should be representative of its capabilities. Eventually, the JOVIAL verifier should be able to mechanically duplicate the manipulations performed in this hand-generated scenario.

C. Verifier Development

The JOVIAL verifier will be developed in two main phases, and will be closely coordinated with the development of key examples. The first version of all components will be completed by 31 December 1979. These components will then be integrated to form the first version of the complete verifier by 31 March 1980. This verifier will then be

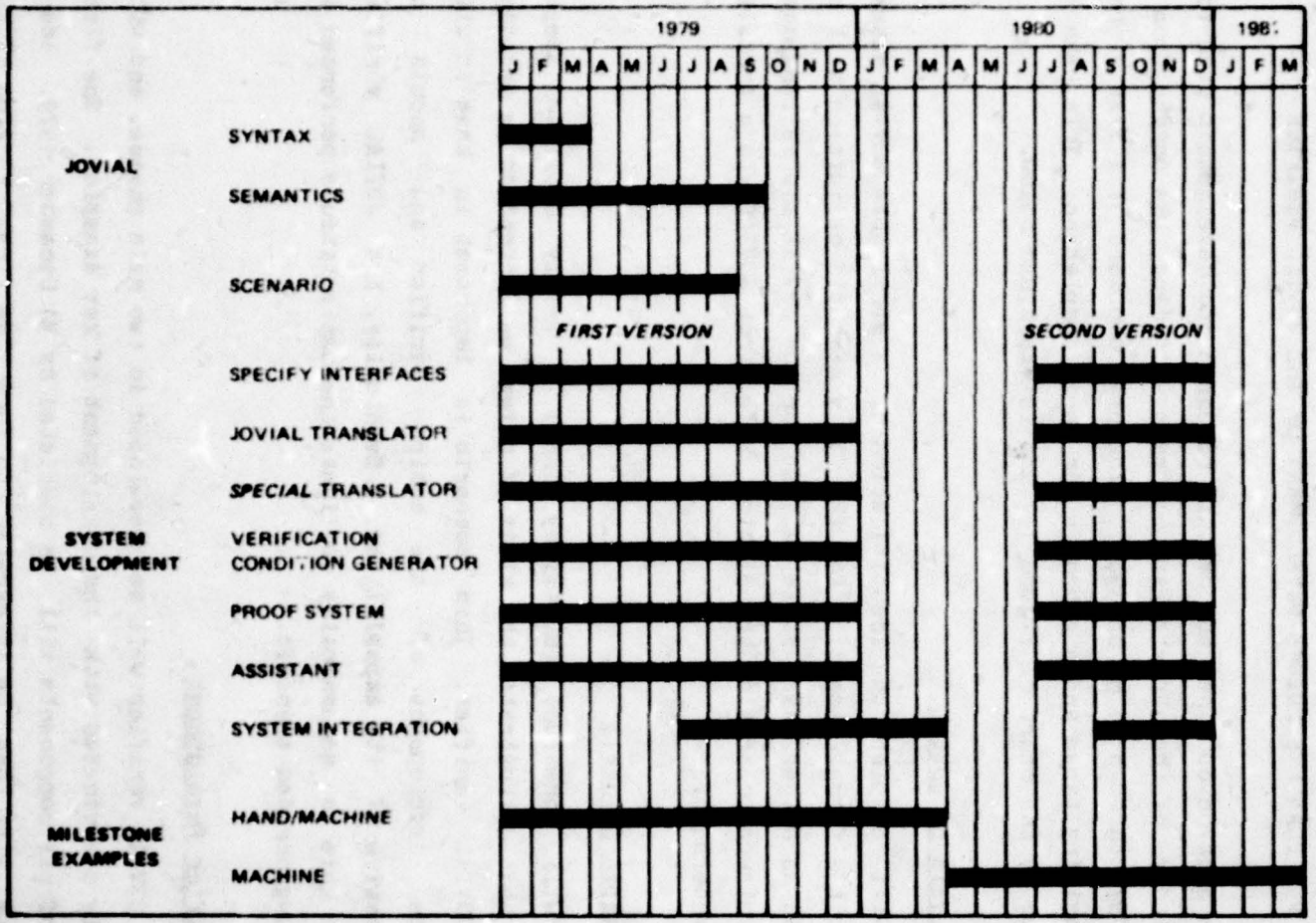


FIGURE V-1 RESEARCH AND DEVELOPMENT PLAN

exercised on examples for three months, at which time we naturally expect certain now unforeseen problems to surface.

Inadequacies found in the first version of the verifier will be removed (if theoretically and practically possible) in the second version of the verifier, which will be developed during the six-month period from 1 July 1980 through 31 December 1980. This second version of the verifier will then be used in completing the design and verification of our two milestone examples.

D. Milestone Examples

Perhaps the main overall characteristic of our plan is the example-driven nature of the verifier's development. We have taken it to be axiomatic that early attention to real examples is critical if we are to produce an effective JOVIAL verifier. To a large extent, we expect problems encountered while attempting to verify certain real programs to drive the entire technical direction of the verifier.

Consequently, as explained in Section IV, we carefully chose to consider (as yet undetermined) critical parts of a secure operating system and a fault-tolerant avionics computer. Work on these examples will begin in January 1980. Attention will initially focus on analyzing the appropriateness of their design and specification, and on making any adjustments needed for verification purposes. The verification of both examples will begin as soon as possible, with the JOVIAL verifier being used to an increasingly greater degree as its development progresses.

VI SUMMARY

This report has described the progress made during the period from 1 November 1977 through 31 December 1978, regarding:

- The analysis, enhancement, and formal definition of the JOVIAL programming language (J73/I)
- The organization, technical development, and implementation of the JOVIAL verifier
- The selection, analysis, and verification of two real systems of significant size and complexity, critical parts of which will be written in JOVIAL and verified by the JOVIAL verifier.

A comprehensive plan for completing each remaining task has also been described. This plan calls for a considerable amount of work in each of these three areas during the coming year, including the completion of the initial version of all components of the JOVIAL verifier by 31 December 1979.

REFERENCES

1. R. S. Boyer and J S. Moore, A Computational Logic, to appear in the ACM Monograph Series published by Academic Press.

2. R. S. Boyer and J S. Moore, "A Computer Proof of the Correctness of a Simple Optimizing Compiler for Expressions", Computer Science Laboratory Technical Report 5, SRI Project 4079, Stanford Research Institute, Menlo Park, CA (1977).

3. R. S. Boyer and J S. Moore, "A Fast String Searching Algorithm," CACM, Vol. 20, No. 10, pp. 762-772 (October 1977).

4. R. S. Boyer and J S. Moore, "A Formal Semantics for the SRI Hierarchial Program Design Methodology," Computer Science Laboratory Report, SRI International Menlo Park, CA (November 1978).

5. R. S. Boyer and J S. Moore, "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory," Proceedings of the Fifth International Conference on Artificial Intelligence, Vol. 1, pp. 511-519 (August 1977).

6. L. P. Deutsch, "An Interactive Program Verifier," Ph.D. thesis, University of California at Berkeley (1973); also available as Xerox Palo Alto Research Center Report CSL-73-1 (May 1973).

7. E. W. Dijkstra, A Discipline of Programming, (Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976).

8. B. Elspas, K.N. Levitt, and R.J. Waldinger, "An Interactive System for the Verification of Computer Programs," Final Report, SRI Project 1891, Stanford Research Institute, Menlo Park, CA (September 1973).

9. B. Elspas, R. S. Boyer, R. E. Shostak, and J. M. Spitzen, "A Verification System for JOVIAL/J3 Programs (Rugged Programming Environment--RPE/1)," Computer Science Laboratory Report CSL-55, Project 3756, Stanford Research Institute, Menlo Park, CA (January 1976).

10. B. Elspas, R. E. Shostak, and J. M. Spitzen, "A Verification System for JOCIT/J3 Programs (Rugged Programming Environment--RPE/2)," Final Report, SRI Project 5042, Stanford Research Institute, Menlo Park, CA (April 1977).

11. B. Elspas and R. E. Shostak, "The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness," Final Report, SRI Project 2686, SRI International, Menlo Park, CA (August 1978).

12. D. I. Good, R. L. London, and W. W. Bledsoe, "An Interactive Program Verification System," IEEE Trans. on Software Engineering, SE-1, pp. 59-67 (March 1975).

13. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," C.ACM, Vol. 12, No. 10, pp. 576-583 (October 1969).

14. JOVIAL J73/I Computer Programming Manual, Computer Sciences Corporation, 650 North Sepulveda Blvd., El Segundo, CA (October 1977).

15. J. C. King, "A Program Verifier," Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA (1969).

16. R. L. London, et al., "Proof Rules for the Programming Language EUCLID," Acta Informatica, Vol. 10, pp. 1-26 (1978).

17. D. C. Luckham, "Program Verification and Verification-Oriented Programming," Proceedings of IFIP Congress 77, North-Holland Publishing Co., 1977, pp. 783-793.

18. Military Standard, JOVIAL (J73/I), MIL-STD-1589 (USAF) Department of the Air Force (28 February 1977).

19. M. S. Moriconi, "A Designer/Verifier's Assistant," Computer Science Laboratory Report CSL-80, SRI International, Menlo Park, CA (October 1978); also IEEE Trans. on Software Eng. (to appear).

20. M. S. Moriconi, "A System for Incrementally Designing and Verifying Programs," Ph.D. thesis, University of Texas at Austin (December 1977); also available as Computer Science Laboratory Reports CSL-73 and CSL-74, Menlo Park, CA.

21. P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, "A Provably Secure Operating System: The System, Its Applications, and Proofs," Computer Science Laboratory Final Report, Project 4332 (February 1978).

22. L. Robinson, "HDM--Command and Staff Overview," Computer Science Laboratory Report CSL-49, SRI International, Menlo Park, CA (February 1978).

23. L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena, "A Formal Methodology for the Design of Operating System Software," in Current Trends in Programming Methodology, Vol. 1, R. T. Yeh, ed. (Prentice-Hall, April 1977).

24. O. Roubine and L. Robinson, "SPECIAL - A Specification and Assertion Language," Computer Science Laboratory Report CSL-46, SRI International, Menlo Park, CA (January 1977).

25. R. E. Shostak, "An Efficient Decision Procedure for Arithmetic with Function Symbols," to appear in J.ACM (tentatively in April 1979).

26. R. E. Shostak, "On the Sup-Inf Method for Proving Presburger Formulas," J.ACM, Vol. 24, No. 4, pp. 529-543 (October 1977).

27. B. A. Silverberg, "A User's Guide to the Interpg Parser Generator System," Computer Science Laboratory Report CSL-78, SRI International, Menlo Park, CA (September 1978).

28. N. Suzuki, "Verifying Programs by Algebraic and Logical Reduction," Proceedings of International Conference on Reliable Software, pp. 473-481 (April 1975).

29. J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: The Design and Analysis of a Fault Tolerant Computer for Aircraft Control," Proc. of the IEEE, pp. 1240-1255 (October 1978).

Appendix A

A FORMAL GRAMMAR FOR JOVIAL-J73/I

1. Tokens of the Grammar:

```
B  BC BEGIN  BIT_BYTE BLOCK  BY CHAR COMPOOL COPY DEF
DEFAULT DEFINE DIGIT DOUBLE E  EJECT ELSE END F FLOAT
FOR GOTO IF ITEM LETTER LINKAGE LIST LOC LOGOP M MARK
NMD NOLIST NOT NUMBER NUMOP OTHERCHARACTER OVERLAY  P
PROC PROGRAM  REDUCIBLE RELOP RETURN  ROLLBACK  SHIFT
SIGN SIZEKEY SKIP STATUS STOP STRING SU SWITCH SYMBOL
TABLE THEN TR TRACE V WHILE
```

2. Modified BNF Grammar

```
/*  1*/ compilation          : compil
                             | compilation compil ;

/*  3*/ compil               : dirplusq
                                  compool_program_procdecl ;

/*  4*/ dirplusq             :
                             | dirplus ;

/*  6*/ dirplus              : dir
                             | dirplus dir ;

/*  8*/ dir                  : '!' dir_body ;
```

```
/*  9*/ dir_body            : COMPOOL compool_body ';'
                            | SKIP letterq ';'
                            | BEGIN ';'
                            | END ';'
                            | TRACE bit_formulaq namelist ';'
                            | COPY char_const ';'
                            | LIST list_control ';'
                            | EJECT ';'
                            | NOLIST list_control ';'
                            | LINKAGE symbols ';'
                            | DOUBLE namelist ';'
                            | ROLLBACK ';'
                            | REDUCIBLE ';' ;

/* 22*/ bit_formulaq         :
                            | bit_formula ;

/* 24*/ letterq              :
                            | LETTER ;

/* 26*/ list_control         :
                            | 'DEFINE/COPY' ;

/* 28*/ symbols              : symbol
                            | symbols symbol ;

/* 30*/ symbol               : SYMBOL
                            | SIGN
                            | number
                            | const
                            | comment
                            | define_call ;

/* 36*/ name                 : SYMBOL ;
```

```
/* 37*/ define_call          : name a_defpqsq ;

/* 38*/ a_defpqsq            :
                             | a_defpqs ;

/* 40*/ a_defpqs             : a_defpq
                             | a_defpqs ',' a_defpq ;

/* 42*/ a_defpq              :
                             | STRING ;

/* 44*/ compool_body         : constq names2
                             | '(' constq ')' ;

/* 46*/ comment              : STRING ;

/* 47*/ integer              : number
                             | bit_const
                             | qual_status_const
                             | ''' character ''' ;

/* 51*/ constq               : const
                             | ;

/* 53*/ const                : numeric_const
                             | bit_const
                             | char_const
                             | CHAR '(' const ')' ;

/* 57*/ numeric_const        : integer_const
                             | float_const
                             | FLOAT '(' const ')' ;

/* 60*/ bit_const            : 'ONE:FIVE' B ''' beads ''' ;

/* 61*/ beads                : bead
                             | beads bead ;
```

```
/* 63*/ char_const          : ''' chars ''' ;

/* 64*/ chars               : character
                            | chars character ;

/* 66*/ character           : LETTER
                            | DIGIT
                            | MARK
                            | OTHERCHARACTER ;

/* 70*/ bead                : DIGIT
                            | 'LETTER/A:V' ;

/* 72*/ integer_const       : number
                            | status_const
                            | qual_status_const ;

/* 75*/ qual_status_const   : V '(' name ':' SYMBOL ')' ;

/* 76*/ number              : NUMBER
                            | '(' number_formula ')' ;

/* 78*/ number_formula      : integer
                            | 'PLUS/MINUS/NOT' number_formula
                            | number_formula 'OP.NOT**'
                                 number_formula
                            | SHIFT '(' number_formula ','
                                number_formula ')'
                            | 'ABS/SGN' number_formula
                            | 'BITOF/INT' '(' numform_const ')'
                            | '(' number_formula ')'
                            | 'MACHINE.PARAMETER'
                            | size_funcall ;
```

```
/* 87*/ numform_const          : number_formula
                               | const ;

/* 89*/ size_funcall           : SIZEKEY '(' size_arg ')' ;

/* 90*/ size_arg               : formula
                               | name ;

/* 92*/ float_const            : float_const1
                               | float_const2 ;

/* 94*/ float_const1           : NUMBER E signq NUMBER
                                     precisionq ;

/* 95*/ signq                  : 'PLUS/MINUS'
                               | ;

/* 97*/ precisionq             :
                               | M plusq NUMBER ;

/* 99*/ plusq                  :
                               | '+' ;

/*101*/ float_const2           : float_prfx exponq precisionq ;

/*102*/ float_prfx             : NUMBER
                               | numberq '.' NUMBER ;

/*104*/ numberq                :
                               | NUMBER ;

/*106*/ exponq                 :
                               | E signq NUMBER ;
```

```
/*108*/ status_consts          : status_const
                               | status_consts ','
                                     status_const ;

/*110*/ status_const           : V '(' SYMBOL ')' ;

/*111*/ statuslist_decl        : STATUS name o_s_integerq
                                  status_consts status_bodylq
                                  ';' ;

/*112*/ status_bodylq          :
                               | status_bodyl ;

/*114*/ status_bodyl           : status_body
                               | status_bodyl status_body ;

/*116*/ status_body            : '[' o_s_integer ']'
                                     status_consts ;

/*117*/ names2                 : name2
                               | names2 name2 ;

/*119*/ name2                  : name
                               | '(' name ')' ;

/*121*/ compool_program_procdecl : compool
                               | program
                               | proc_decl ;

/*124*/ compool                : COMPOOL name ';'
                                     cmpl_source_body ;

/*125*/ cmpl_source_body       : compool_decl ';'
                               | BEGIN compool_decls END ;
```

```
/*127*/ compool_decls          : compool_decl
                             | compool_decls compool_decl
                             | ;

/*130*/ compool_decl           : overlay_decl
                             | data_decl
                             | name_decl
                             | proc_decl
                             | define_decl
                             | statuslist_decl
                             | def_decl ;

/*137*/ data_decl              : item_decl
                             | table_decl
                             | block_decl ;

/*140*/ item_decl              : ITEM name alloc_specq idesc
                                   idecl_tailq ;

/*141*/ alloc_specq            :
                             | 'IN/RESERVE'
                             | '@' nameq ;

/*144*/ idecl_tailq            :
                             | c_s_const
                             | LOC '(' loc_arg ')' ;

/*147*/ loc_arg                : name
                             | scalar_var ;

/*149*/ name_decl              : name namelist ';' ;

/*150*/ block_decl             : BLOCK name alloc_specq ';'
                                   block_body ;
```

```
/*151*/ block_body          : ';'
                            | data_decl
                            | BEGIN block_body2sq END ;

/*154*/ block_body2sq       :
                            | block_body2s ;

/*156*/ block_body2s        : block_body2
                            | block_body2s block_body2 ;

/*158*/ block_body2         : ';'
                            | data_decl
                            ; overlay_decl
                            | statuslist_decl
                            | define_decl ;

/*163*/ table_decl          : ord_table_decl
                            | spec_table_decl ;

/*165*/ ord_table_decl      : TABLE name alloc_specq
                              dimension_list structure
                              packingq ord_table_tail ;

/*166*/ dimension_list      : '[' dimensions ']' ;

/*167*/ dimensions          : lower_dimq o_s_integer
                            | dimensions ',' lower_dimq
                                 o_s_integer ;

/*169*/ lower_dimq          :
                            | o_s_integer ':' ;

/*171*/ structure           :
                            | P ;
```

```
/*173*/ packingq              :
                              | NMD ;

/*175*/ ord_table_tail        : idesc constantlistq ';'
                              | constantlistq ';'
                                    ord_table_body ;

/*177*/ ord_table_body        : ';'
                              | ord_table_idecl
                              | BEGIN ord_table_body_declsq
                                    END ;

/*180*/ ord_table_body_declsq :
                              | ord_table_body_decls ;

/*182*/ ord_table_body_decls  : ord_table_body_decl
                              | ord_table_body_decls
                                    ord_table_body_decl ;

/*184*/ ord_table_body_decl   : ord_table_idecl
                              | statuslist_decl
                              | define_decl ;

/*187*/ ord_table_idecl       : ITEM name idesc packingq
                                    constantlistq ';' ;

/*188*/ constantlistq         :
                              | constantlist ;

/*190*/ constantlist          : const_indxlq const_list_el
                                  const_list_tailsq ;

/*191*/ const_indxlq          :
                              | '[' const_indxl ']' ;
```

```
/*193*/ const_indxl             : c_s_integer
                                | const_indxl ',' c_s_integer ;

/*195*/ const_list_el           : const_list_el1
                                | const_list_el ','
                                      const_list_el1 ;

/*197*/ const_list_el1          : c_s_const
                                | LOC '(' loc_arg ')'
                                | number '(' const_list_elq ')'
                                | ;

/*201*/ const_list_elq          :
                                | const_list_el ;

/*203*/ const_list_tailsq       :
                                | const_list_tails ;

/*205*/ const_list_tails        : const_indxl const_list_el
                                | const_list_tails const_indxl
                                    const_list_el ;

/*207*/ spec_table_decl         : TABLE name alloc_specq
                                  dimension_list structure
                                  NUMBER spec_table_decl_tail ;

/*208*/ spec_table_decl_tail    : idesc packingq '[' number
                                    cnumberq ']' constantlistq ';'
                                | constantlistq ';'
                                    spec_table_body ;

/*210*/ spec_table_body         : ';'
                                | spec_table_idecl
                                | BEGIN spec_table_body_declsq
                                    END ;
```

```
/*213*/ spec_table_idecl          : ITEM name idesc packingq
                                    '[' cnumberq ']'
                                    constantlistq ';' ;

/*214*/ spec_table_body_declsq    :
                                  | spec_table_body_decls ;

/*216*/ spec_table_body_decls     : spec_table_body_decl
                                  | spec_table_body_decls
                                      spec_table_body_decl ;

/*218*/ spec_table_body_decl      : spec_table_idecl
                                  | statuslist_decl
                                  | define_decl ;

/*221*/ overlay_decl              : OVERLAY overlay_itemq
                                      overlay_expr ';' ;

/*222*/ overlay_itemq             :
                                  | '[' number_bitconst ']' ;

/*224*/ number_bitconst           : number
                                  | bit_const ;

/*226*/ overlay_expr              : overlay_string
                                  | overlay_expr ':'
                                      overlay_string ;

/*228*/ overlay_string            : overlay_element
                                  | overlay_string ','
                                      overlay_element ;
```

```
/*230*/ overlay_element          : number
                                 | name
                                 | '(' overlay_expr ')' ;

/*233*/ external_decl            : 'DEF/REF' external_decl_body ;

/*234*/ external_decl_body       : ext_body_decl
                                 | BEGIN ext_body_declsq END ;

/*236*/ ext_body_decl            : ';'
                                 | data_decl
                                 | name_decl
                                 | proc_decl
                                 | define_decl
                                 | statuslist_decl ;

/*242*/ ext_body_declsq          :
                                 | ext_body_decls ;

/*244*/ ext_body_decls           : ext_body_decl
                                 | ext_body_decls ext_body_decl ;

/*246*/ define_decl              : DEFINE name f_defplq
                                     define_string ';' ;

/*247*/ define_string            : STRING ;

/*248*/ f_defplq                 :
                                 | '(' f_defpl ')' ;

/*250*/ f_defpl                  : LETTER
                                 | f_defpl ',' LETTER ;

/*252*/ def_decl                 : DEF def_decl_body ;
```

```
/*253*/ def_decl_body          : data_decl
                               | BEGIN def_decl2s END ;

/*255*/ def_decl2s             : def_decl2
                               | def_decl2s def_decl2 ;

/*257*/ def_decl2              : data_decl
                               | define_decl
                               | statuslist_decl
                               | def_decl
                               | ;

/*262*/ program                : PROGRAM name ';' stat_decl ;

/*263*/ proc_decl              : proc_clause procdirplusq
                                   stat_decl ;

/*264*/ proc_clause            : PROC name data_allocq
                                   fioplq ';' ;

/*265*/ procdirplusq           : procdirplus
                               | ;

/*267*/ procdirplus            : procdir
                               | procdirplus procdir ;

/*269*/ procdir                : '!' TRACE bit_formulaq
                                   namelist ';'
                               | '!' REDUCIBLE ';'
                               | '!' LINKAGE symbols ;

/*272*/ data_allocq            : 'IN/RESERVE/AT'
                               | ;
```

```
/*274*/ fioplq                  : proc_fiopl
                                | fn_fipl ;

/*276*/ proc_fiopl              : '(' fipsq ':' fops ')' ;

/*277*/ fn_fipl                 : '(' namelist ')' idescoq ;

/*278*/ idescoq                 :
                                | idesc idesc_tail ;

/*280*/ idesc                   : BC numberq
                                | F trq numberq cnumberq
                                | SU numberq nameq ;

/*283*/ cnumberq                :
                                | ',' NUMBER ;

/*285*/ trq                     :
                                | ',' TR ;

/*287*/ idesc_tail              :
                                | c_s_const ;

/*289*/ c_s_const               : const
                                | 'PLUS/MINUS' const ;

/*291*/ fipsq                   : namelist
                                | ;

/*293*/ namelist                : name
                                | namelist ',' name ;

/*295*/ fops                    : namelist ;
```

```
/*296*/ stat_decl          : statement
                         | decl ;

/*298*/ statement          : stat
                         | simple_if
                         | name ':' assertq statement ;

/*301*/ decl               : assert_decl
                         | data_decl
                         | statuslist_decl
                         | define_decl
                         | name_decl
                         | proc_decl
                         | external_decl
                         | BEGIN decls END ;

/*309*/ decls              : decl
                         | decls decl ;

/*311*/ assert_decl        : 'ASSERTIN/OUT' formula ;

/*312*/ stat               : simple_stat
                         | compd_stat
                         | name ':' assertq stat ;

/*315*/ simple_stat        : ';'
                         | assign_stat
                         | goto_stat
                         | return_stat
                         | stop_stat
                         | loop_stat
                         | if_stat2
                         | switch_stat
                         | proccall_stat
                         | assert_stat ;
```

```
/*325*/ simple_if          : IF bit_formula ';' statement ;

/*326*/ if_stat2           : IF bit_formula ';' stat
                               ELSE statement ;

/*327*/ compd_stat         : BEGIN stat_declplusq
                               labelsq END ;

/*328*/ stat_declplusq     :
                           | stat_declplus ;

/*330*/ stat_declplus      : stat_decl
                           | stat_declplus stat_decl ;

/*332*/ labelsq            :
                           | labels ;

/*334*/ labels             : name ':'
                           | labels name ':' ;

/*336*/ assign_stat        : vars '=' formula ;

/*337*/ goto_stat          : GOTO name ';' ;

/*338*/ return_stat        : RETURN nameq ';' ;

/*339*/ stop_stat          : STOP ';' ;

/*340*/ switch_stat        : SWITCH numeric_formula ';'
                               switch_tailq ;

/*341*/ switch_tailq       : BEGIN sw_body labelsq END ;

/*342*/ names              : name
                           | names name ;
```

```
/*344*/ sw_body                 : sw_body1
                                | sw_body sw_body1 ;

/*346*/ sw_body1                : '[' sw_listq ']' statement
                                    commaq ;

/*347*/ commaq                  :
                                | ',' ;

/*349*/ sw_listq                :
                                | sw_list ;

/*351*/ sw_list                 : sw_list1
                                | sw_list ',' sw_list1 ;

/*353*/ sw_list1                : DEFAULT
                                | c_s_integer os_int_tailq ;

/*355*/ c_s_integerq            :
                                | '[' c_s_integer ']' ;

/*357*/ c_s_integer             : integer
                                | 'PLUS/MINUS' integer ;

/*359*/ os_int_tailq            :
                                | c_s_integer ;

/*361*/ loop_stat               : while_stat
                                | for_stat ;

/*363*/ while_stat              : WHILE assertq bit_formula
                                    ';' statement ;

/*364*/ for_stat                : FOR name ':' assertq
                                    controls ';' statement ;
```

```
/*365*/ assertq                 :
                                | assert_stat ;

/*367*/ controls                : controls2q
                                | formula controls2q ;

/*369*/ controls2q              :
                                | BY numeric_formula while_formq
                                | THEN formula while_formq
                                | WHILE bit_formula by_thenq ;

/*373*/ while_formq             :
                                | WHILE bit_formula ;

/*375*/ by_thenq                :
                                | BY numeric_formula
                                | THEN formula ;

/*378*/ assert_stat             : 'ASSERT/ASSUME/PROVE'
                                    formula ';' ;

/*379*/ nameq                   :
                                | name ;

/*381*/ vars                    : var
                                | vars ',' var ;

/*383*/ var                     : named_var
                                | function_var ;

/*385*/ named_var               : scalar_var
                                | indexed_var ;

/*387*/ scalar_var              : name base_formq ;
```

```
/*388*/ base_formq            :
                              | '@' numeric_formula ;

/*390*/ indexed_var           : name '[' indices ']'
                                  base_formq ;

/*391*/ indices               : numeric_formula
                              | indices ',' numeric_formula ;

/*393*/ function_var          : BIT_BYTE '(' named_var ','
                                  numeric_formula tailq ')' ;

/*394*/ tailq                 :
                              | ',' numeric_formula ;

/*396*/ formula               : char_formula
                              | bit_formula
                              | numeric_formula ;

/*399*/ numeric_formula       : numeric_const
                              | var
                              | funcall
                              | bit_formula
                              | 'PLUS/MINUS' numeric_formula
                              | numeric_formula NUMOP
                                  numeric_formula
                              | '(' numeric_formula ')' ;

/*406*/ char_formula          : char_const
                              | char_var
                              | funcall
                              | '(' char_formula ')' ;

/*410*/ char_var              : named_var
                              | function_var ;
```

```
/*412*/ bit_formula          : bit_const
                             | bit_var
                             | funcall
                             | logical_formula
                             | rel_formula
                             | '(' bit_formula ')'
                             | numeric_formula
                             | char_formula ;

/*420*/ bit_var              : named_var
                             | indexed_var
                             | function_var ;

/*423*/ logical_formula      : bit_formula LOGOP bit_formula
                             | NOT bit_formula
                             | SHIFT '(' bit_formula ','
                                 numeric_formula bitfn_tailq
                                 ')' ;

/*426*/ bitfn_tailq          :
                             | ',' numeric_formula ;

/*428*/ rel_formula          : formula RELOP formula ;

/*429*/ data_baseq           :
                             | '@' numeric_formula ;

/*431*/ proccall_stat        : name data_baseq a_ioplq ';' ;

/*432*/ funcall              : name data_baseq '(' a_ipsq ')' ;

/*433*/ a_ioplq              :
                             | '(' a_iopl ')' ;
```

```
/*435*/ a_iopl          : a_ips
                       | a_ipsq ':' a_opl ;

/*437*/ a_opl           : a_op
                       | a_opl ',' a_op ;

/*439*/ a_ips           : a_ip
                       | a_ips ',' a_ip ;

/*441*/ a_ip            : name data_baseq
                       | formula
                       | '@' numeric_formula ;

/*444*/ a_ipsq          :
                       | a_ips ;

/*446*/ a_op            : var ;
```

3. Terminal Symbols and Metaterminals of the Grammar

! ' ( ) * , . : ; = @ ABS/SGN ASSERT/ASSUME/PROVE ASSERTIN/OUT B BC BEGIN BITOF/INT BIT_BYTE BLOCK BY CHAR COMPOOL COPY DEF DEF/REF DEFAULT DEFINE DEFINE/COPY DIGIT DOUBLE E EJECT ELSE END F FLOAT FOR GOTO IF IN/RESERVE IN/RESERVE/AT ITEM LETTER LETTER/A:V LINKAGE LIST LOC LOGOP M MACHINE.PARAMETER MARK NMD NOLIST NOT NUMBER NUMOP ONE:FIVE OP.NOT** OTHERCHARACTER OVERLAY P PLUS/MINUS PLUS/MINUS/NOT PROC PROGRAM REDUCIBLE RELOP RETURN ROLLBACK SHIFT SIGN SIZEKEY SKIP STATUS STOP STRING SU SWITCH SYMBOL TABLE THEN TR TRACE V WHILE [ ]

4. Definitions of the Metaterminals (TYPEOFLIST)

The first element of each parenthesized item is the metaterminal, and the following elements are its alternative instances. Thus SU stands for an S or a U.

```
(SU S U)
(ASSERTIN/OUT ASSERTIN ASSERTOUT)
(BC B C)
(TR T R)
(DEF/REF DEF REF)
(BIT/BYTE BIT BYTE)
(IN/RESERVE/AT IN RESERVE @)
(RELOP = < > <= >= <>)
(LOGOP AND OR XOR EQV)
(NUMOP + - * / \ **)
(OP.NOT** + - * / \ = <> <= >= < > AND OR XOR EQV)
(PLUS/MINUS + -)
(ONE:FIVE 1 2 3 4 5)
(ASSERT/ASSUME/PROVE ASSERT ASSUME PROVE)
(DEFINE/COPY DEFINE COPY)
(SIGN + - * / \ ** = < > <= >= <> , : ; ! ( ) [ ] @
      NOT AND OR XOR EQV BEGIN END)
(PLUS/MINUS/NOT + - NOT)
(ABS/SGN ABS SGN)
(BITOF/INT BITOF INT)
(IN/RESERVE IN RESERVE)
(SIZEKEY BITSIZE BYTESIZE WORDSIZE)
(MACHINE.PARAMETER BITSINBYTE BITSINWORD LOCSINWORD
      BYTESINWORD ADDRESSSIZE)
(NMD N M D)
(LETTER A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)
(DIGIT 0 1 2 3 4 5 6 7 8 9)
(MARK + - * / \ > < = @ . : , ; ( ) [ ] ' " ! $)
(OTHERCHARACTER    !)
(LETTER/A:V A B C D E F G H I J K L M N O P Q R S T U V)
```

Appendix B

AN AXIOMATIZATION OF J73/I REAL ARITHMETIC

1. Preliminaries

In order to produce correctness proofs for J73/I programs that employ floating-point (real) arithmetic we need to consider two aspects of arithmetic axiomatization.

(1) The semantics of input and output of representations of numerical values.

(2) The semantics of arithmetic operations acting on internal machine representations of numbers.

A J73/I program may obtain input values either by accessing a previously stored result or by accepting input from a device such as a display terminal. Similarly, outputs of a program may be stored internally for future use or transmitted to an output device, or both. Since humans like to express themselves in decimal notation with as few formatting restrictions as possible, conversion routines are usually called upon to convert input data into a standard internal machine representation, or the latter into output data.

As a result, one cannot assume that the "value" of a program variable is mathematically equal to the value given as input. Nor can one even assume that input and output are inverse operations. For example, the decimal number 0.1 is not exactly representable in the usual implementations of floating-point numbers on binary machines.

Moreover, a program that does nothing but accept a number and then print it will usually accept, say, 3.1415926535897932, and then print some truncated or rounded value such as 3.1415927.

The semantics of I/O conversion can be cleanly separated from the semantics of arithmetic operations by insisting that all program assertions (and theorems) that refer to named program variables, including input and output variables, are interpreted as statements about actual values that are <u>exactly</u> machine representable. By taking this viewpoint, I/O conversion details become a secondary issue that can be dealt with once and for all by proving properties of the conversion programs.

Note that we did not require in the above interpretation that literal constants appearing in assertions be machine representable. It might be perfectly correct to assert that

$$x + y < 0.1$$

even though 0.1 is not machine representable. On the other hand, an assertion that

$$x + y = 0.1$$

might make mathematical sense in the context of the real domain but not be true of any pair of values in the machine representable domain of floating-point numbers (henceforth called R). For the latter assertion to make sense in most contexts, one would instead write

$$x + y = Rep(0.1)$$

where Rep(v) is a function that returns the unique value contained in the domain R that is "closest" to the real value v. One would hope that an input device accepting a value (say, z) for input to a program would set the actual value of z to Rep(input) but one cannot be sure that this will occur because the semantics of J73/I do not apply to I/O devices. A precise definition of Rep(v) will be given in Section 3 of this appendix.

2. Choosing an Abstract Representation for R

In selecting a mathematically clean abstract representation for machine numbers and arithmetic we are necessarily influenced by what actual implementations of J73/I do (or should do according to the informal semantics). In particular, numbers are of a few fixed lengths, have fixed size ranges, and are manipulated by hardware arithmetic units that have what we might call "customary implementations". The capabilities of the hardware in effect determine the semantics of most of the arithmetic operations. We want to model what a good customary implementation should do rather than cater to a variety of possible idiosyncracies that might occur in a poor implementation of arithmetic on some particular machine.

A possible representation of numbers in R would be an indexed array of digits where each digit d satisfies $0 <= d < b$ (where b is the number base of the implementation), and the exponent multiplier is determined by the smallest index value for which an array value is nonzero. For example, the octal number 2.35×8^{-6} might be represented by an array A where A[-6] = 2, A[-5] = 3, A[-4] = 5, and all other values of the array are zero. One could then describe all of the effects of arithmetic statements in a program in terms of exact integer arithmetic on arrays. This approach has some attractive features but would probably produce too much low-level detail to be handled effectively in the verification system.

Another way to proceed (which is closer to the customary implementations) is to represent a number v in R by a pair of integers (n,e) such that

$$v = n * b^e$$

where b, (b > 1) is the integer base of the number system. With this representation the numbers in R form an algebraic ring, so that if u and v are in R, then the values u+v, u-v, u*v, -v, abs(v) are also in R. Moreover, arithmetic using these operators is associative and distributive.

Unfortunately, the finite size of machine representations imposes additional restrictions of the form:

$$N1 \leq n \leq N2$$

$$E1 \leq e \leq E2 .$$

With these restrictions on size of n and e, none of the mathematically exact operations u+v, u-v, u*v, -v, abs(v) necessarily produce numbers contained in R. With u = (n1,e1), v = (n2,e2), and u,v both in R, the product u*v given exactly by

$$(n1 * n2) * b^{(e1+e2)}$$

can fail to meet either or both of the above constraints. If the constraint on size of (e1+e2) is violated (exponent over/underflow), then there is no value in R that is "close" to u*v, and Rep(u*v) does not exist. By "close", we mean the following:

Let $z = (n,e) = n * b^e$ in the domain of <u>unrestricted</u> n and e. Then there is a value in R that is close to z if there exists any pair of values z1 and z2, both in R such that

$$z1 \leq z \leq z2 .$$

Moreover, in this case there will exist two values y1 and y2, both in R such that:

(1) $z1 \leq y1 \leq z \leq y2 \leq z2$.

(2) There exists no value w in R such that $y1 < w < y2$.

One of the values y1, y2 is the closest value to z in R. Condition (2) above merely states that in a total ordering of the values in R there is a closed interval defined by some pair of <u>consecutive</u> values that contains the value z.

Note that we have not required a unique representation of the values in R. Uniqueness of representation will turn out to have no particular significance or importance with respect to properties of the abstract model.

To illustrate the general features of a domain R based on the above representation scheme, consider the particular case of a signed four-decimal-digit "mantissa" with a signed two-decimal-digit exponent. We then have b = 10 and

$$-9999 \leq n \leq 9999$$

$$-99 \leq e \leq 99 .$$

The number 123 can be exactly represented in two ways--e.g.,

$$123 * 10^{0} \quad \text{or} \quad 1230 * 10^{-1} .$$

The number pi = 3.14159265... is not in R, but we have that

$$y1 = 3141 * 10^{-3} < pi < 3142 * 10^{-3} = y2$$

where no value in R lies between y1 and y2. The closest value to pi is in fact y2, and this value happens to have a unique representation in R, although this property will not hold in general.

In all of the following discussion we will assume that reals are internally represented by a signed integer consisting of at most t digits of a fixed base b, associated with a integer exponent e such that

$$E1 \leq e \leq E2 .$$

Except for the semantics of over/underflow it will not be necessary to consider the internal representation of exponents. Exponents could, for example, be scaled to be all non-negative. The values in R are the exact values of the form:

$$n * b^e \quad \text{where} \quad -(b^t - 1) <= n <= b^t - 1 .$$

This differs from customary implementations only in the detail that the above bounds on n assure that if v is in R, then so is -v. This is not necessarily true in some machines--e.g., those using one form of twos-complement arithmetic where the single most-negative machine number has no positive inverse.

In the following section we define some functions of arbitrary real arguments to be used as a basis set for describing precisely the results to be expected from arithmetic operations carried out on values in the domain R.

3. Definitions

It will be useful to define the following functions of a real argument x, recalling that b is the base of the number system and t is the "wordlength". First, the functions Sign, Floor, and Ceiling have the usual meanings. That is,

Sign(x) = if x = 0 then 0 else x/Abs(x)

Floor(x) = largest integer <= x

Ceiling(x) = smallest integer >= x .

Next, we give three completely equivalent definitions of a function Ep(x) which may be used interchangably. Depending on program context, one definition might be more natural or useful than another.

(1) Ep(x) = if x = 0 then 0 else the unique integer p such that

$$b^{t-1} <= Abs(x) * b^p < b^t$$

(2) Ep(x) = if x = 0 then 0

else Ceiling(t - 1 - Log_b(Abs(x)))

(3) $Ep(x) = \text{if } x = 0 \text{ then } 0$

$\text{elsif } Abs(x) < b^t \text{ and } Abs(x) >= b^{t-1} \text{ then } 0$

$\text{elsif } Abs(x) > b^t \text{ then } Ep(x/b) - 1$

$\text{else } Ep(x * b) + 1 .$

We also need the following three functions derived from Ep(x):

$$Bp(x) = b^{Ep(x)}$$

$$Lo(x) = Sign(x) * Floor(Abs(x) * Bp(x)) / Bp(x)$$

$$Hi(x) = Lo(x) + (1 / Bp(x))$$

Among the above functions, Ep(x) effectively returns the number of "shifts" necessary to left-normalize a value of the internal representation. The function Bp(x) is the integer multiplier necessary to accomplish the shift. Lo(x) is the largest value in R that is <= x, and Hi(x) is the smallest value in R that is > x. Note that the definition of Hi(x) requires no special handling of the case where the next value in R after x requires incrementation of the exponent (occurs only when all of the digits of Lo(x) are equal to b-1).

Ignoring for the moment the possibility of exponent over/underflow, one can finally define the function Rep(x) as:

$$Rep(x) = \text{if } x - Lo(x) < Hi(x) - x \text{ then } Lo(x) \text{ else } Hi(x)$$

This is the previously promised function that provides the closest value in R to an arbitrary real x. It is this function that should be applied to values from an input device to obtain the correct internal representation.

4. Axiomatization of Arithmetic

The function Rep(x) derived in the previous section prescribes an unambiguous method of obtaining a "correctly" rounded t-significant digit representation of a real value. Most good machine implementations of arithmetic that use double-length registers in the computation of an arithmetic function of two arguments either round correctly or are capable of doing so. If one decides that the J73/I semantics of arithmetic imply that correct rounding always occurs, then the following equalities are always exact. With u,v in R,

Result of: Abs(u) = Sign(u) * u

Result of: - u = (-1) * u

Result of: u + v = Rep(u + v)

Result of: u - v = Rep(u - v)

Result of: u * v = Rep(u * v)

Result of: u / v = Rep(u / v)

For example, after the J73/I assignment statement

ZZ = (AA + BB) * CC;

the exact value of ZZ is given by

Rep[Rep(AA + BB) * CC].

Recall that Rep is a function that maps arbitrary reals into numbers in the range R. Since actual implementations limit the size of the exponent used in the internal representation, there will be cases where the result of an input operation or an arithmetic operation produces a number with no close representation as defined in Section 2 of this appendix. The actual semantics of operations involving numbers outside the range R has in the past depended on programming language conventions and implementation details. Fairly frequent use has been

made of the convention that a number too small to be represented with a negative exponent in the implemented range is converted to zero, while numbers too large to be represented are converted to the largest machine-representable number. These conversions may or may not be accompanied by run-time error messages. Neither of the above conversion rules is particularly satisfactory from the standpoint of arithmetic axiomatization or, for that matter, actual computational utility.

A much cleaner solution is to insist that if a value z has no close representation in R, then its value is undefined. Moreover, any arithmetic operation taking z as an argument returns the undefined value. To incorporate this rule in the above axiomatization of arithmetic it is only necessary to redefine the function Rep as follows. Let:

Smallest = (The smallest positive non-zero number in R)

Largest = (The largest positive number in R)

$Rep^{\circ}(x)$ = If Abs(x) > Largest or Abs(x) < Smallest

then undefined else Rep(x)

Replacing Rep(x) by $Rep^{\circ}(x)$ in each of the axioms listed above--that is,

Result of: $u \diamond v = Rep^{\circ}(u \diamond v)$

etc.

--ensures that all arithmetic operations are well defined for all arguments and produce either a unique value in R or the undefined value.

5. Error Bounds

For each program statement that carries out one or more arithmetic operations it is possible to state (assert) bounds on either the real value produced or its representation in R. For example, if u, v and x are to be interpreted as real values in a program context, then a program identifier--say, UU--associated with the value of u will have as an actual value either Lo(u) or Hi(u). Therefore a statement such as

$$XX = UU + VV;$$

will have the effect that the actual value of XX satisfies the inequalities

$$Rep[Lo(u) + Lo(v)] \leq XX \leq Rep[Hi(u) + Hi(v)] .$$

There is a sizable literature on the analysis of algorithms, using some form of error bounding mechanism similar to the above. Whether or not this technique can be useful in proving nontrivial properties of programs using substantial amounts of real arithmetic is a matter for future study and experimentation.

[redacted]